

# A Nondeterministic and Abstract Algorithm for Translating Hierarchical Block Diagrams\*

Viorel Preoteasa<sup>1</sup>    Iulia Dragomir<sup>1</sup>    Stavros Tripakis<sup>1,2</sup>

<sup>1</sup>Aalto University, Finland    <sup>2</sup>University of California, Berkeley, USA

November 7, 2016

## Abstract

In this paper we introduce a nondeterministic algorithm for translating hierarchical block diagrams (HBDs) into an abstract algebra of components with three basic composition operations (serial, parallel, and feedback) and with three constants (split, switch, and sink). We prove that despite its internal nondeterminism, the result of the algorithm is deterministic, meaning that all possible algebra expressions that can be generated from a given HBD are equivalent. Then, different determinizations of the algorithm result in different translation strategies which are all semantically equivalent, although each having its pros and cons with respect to various criteria (compositionality, readability, simplifiability, etc.). As an application of our framework, we show how two translation strategies for Simulink introduced in previous work can be formalized as determinizations of the abstract algorithm. We also prove these strategies equivalent, thus answering an open question raised in the earlier work. All results are formalized and proved in Isabelle.

## 1 Introduction

Hierarchical block diagrams (HBDs) are at the heart of several modeling environments for embedded control systems, including the widespread tool Simulink<sup>1</sup>. Being a graphical notation, and in the case of Simulink a “closed” environment in the sense that the tool is not open-source, such diagrams often need to be translated into another formalism which is more amenable to analysis. Several such translations exist, e.g., from Simulink to Lustre [27] primarily for purposes of code generation, and from Simulink to the Refinement Calculus of Reactive Systems [11], primarily for compositional verification.

In this paper we follow the work of [11], which proposes three translation strategies from HBDs to an algebra of components with three basic composition operators: serial, parallel, and feedback. The different strategies are motivated by the fact that each strategy has its own pros and cons. For instance, one strategy may result in shorter and/or easier to understand algebra terms, while another strategy may result in terms that are easier to simplify. A question left open in [11] is whether these translation strategies are *semantically equivalent*, in the sense that they produce semantically equivalent algebra terms, no matter what the original diagram is. This question is answered positively in this paper.

In order to formulate the question precisely, we introduce an *abstract* and *nondeterministic* algorithm for translating HBDs into an abstract algebra of components with three composition operations (serial, parallel, feedback) and three constants (split, switch, and sink). By *abstract algorithm* we understand an algorithm that produces terms in this abstract algebra. Concrete versions for this algorithm are obtained when using it for concrete models of the algebra (e.g., *constructive functions*). The algorithm is *nondeterministic* in

---

\*This work has been partially supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).

<sup>1</sup><http://www.mathworks.com/products/simulink/>

the sense that it consists of a set of basic operations (transformations) that can be applied in any order. This allows to capture various deterministic translation strategies as determinizations (*refinements* [4]) of the abstract algorithm.

The main results of the paper are the following. (1) We prove that despite its internal nondeterminism, the result of the abstract algorithm is deterministic in the sense that all possible algebra terms that can be generated by the different nondeterministic choices are semantically equivalent. (2) We formalize two of the translation strategies introduced in [11], namely the *feedback-parallel* and *incremental* translation, as refinements of the abstract algorithm. (3) We prove the equivalence of these two translation strategies. (The third strategy introduced in [11], called *feedbackless*, can also be formalized and proved equivalent in our framework; this discussion is omitted here due to lack of space and will be reported in an extended version of this paper.)

All results have been formalized and proved in the Isabelle theorem prover [19] and are available at <http://rcrs.cs.aalto.fi/abstract-translation.zip>.

## 2 Related Work

Model transformation and the verification of its correctness is a long standing line of research, which includes classification of model transformations [3] and the properties they must satisfy with respect to their intent [15], verification techniques [1], frameworks for specifying model transformations (e.g., ATL [12]), and various implementations for specific source and target meta-models. Extensive surveys of the above can be found in [3, 7, 1].

Several translations from Simulink have been proposed in the literature, including to Hybrid Automata [2], BIP [25], NuSMV [17], Lustre [27], Boogie [22], Timed Interval Calculus [8], Function Blocks [28], I/O Extended Finite Automata [29], Hybrid CSP [30], and SpaceEx [18]. It is unclear to what extent these approaches provide formal guarantees on the determinism of the translation. For example, the order in which blocks in the Simulink diagram are processed might a-priori influence the result. Some works fix this order, e.g., [22] compute the control flow graph and translate the model according to this computed order. In contrast, we prove that the results of our algorithm are equivalent for any order.

The focus of several works is to validate the preservation of the semantics of the original diagram by the resulting translation (e.g., see [28, 23, 6, 24]). In contrast, our goal is to prove equivalence of all possible translations. Given that Simulink semantics is informal (“what the simulator does”) ultimately the only way to gain confidence that the translation conforms to the original Simulink model is by simulation (e.g., as in [11]).

With respect to the target algebra of our translation, the most relevant related works are the algebra of flownomials [26] and the relational model for non-deterministic dataflow [14]. A comparison with these works is presented in Section 5.

## 3 Preliminaries

For a type or set  $X$ ,  $X^*$  is the type of finite lists with elements from  $X$ . We denote the empty list by  $\epsilon$ ,  $(x_1, \dots, x_n)$  denotes the list with elements  $x_1, \dots, x_n$ , and for lists  $x$  and  $y$ ,  $x \cdot y$  denotes their concatenation. The length of a list  $x$  is denoted by  $|x|$ . The list of common elements of  $x$  and  $y$  in the order occurring in  $x$  is denoted by  $x \cap y$ . The list of elements from  $x$  that do not occur in  $y$  is denoted by  $x \ominus y$ . We define  $x \oplus y = x \cdot (y \ominus x)$ , the list of  $x$  concatenated with the elements of  $y$  not occurring in  $x$ . A list  $x$  is a *permutation* of a list  $y$  if  $x$  contains all elements of  $y$  (including multiplicities) possibly in a different order.

### 3.1 Constructive Functions

We introduce in this section the *constructive functions* as used in the *constructive semantics* literature [16, 5, 13]. They will provide a concrete model for the abstract algebra of HBDs, introduced in Section 5. These functions are also used in the example from Section 4.

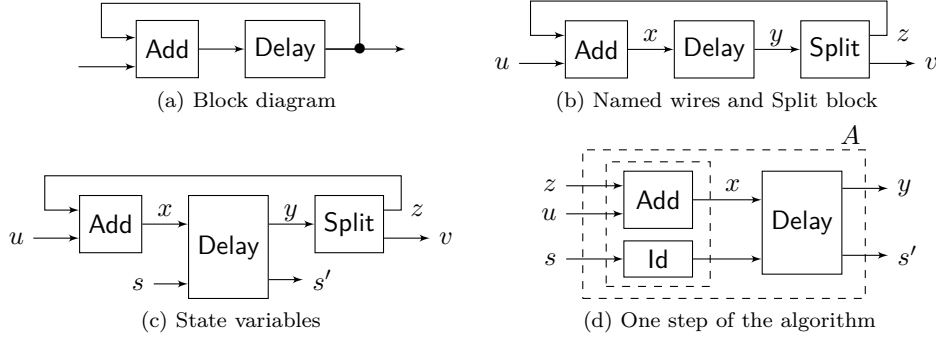


Figure 1: Running example: diagram for summation.

We assume that  $\perp$  is a new element called unknown, and that  $\perp$  is not an element of other sets that we use. For a set  $A$  we define  $A^\perp = A \cup \{\perp\}$ , and on  $A^\perp$  we introduce the *pointed cpo* [9] *partial order* by  $(a \leq b) \iff (a = \perp \vee a = b)$ . We extend the order on  $A^\perp$  to the Cartesian product  $A_1^\perp \times \dots \times A_n^\perp$  by  $(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \iff (\forall 1 \leq i \leq n : x_i \leq y_i)$ .

Constructive functions are the *monotonic* functions  $f : A_1^\perp \times \dots \times A_n^\perp \rightarrow B_1^\perp \times \dots \times B_m^\perp$ , i.e.,  $(\forall x, y : x \leq y \Rightarrow f(x) \leq f(y))$ . We denote these functions by  $A_1 \cdots A_n \xrightarrow{c} B_1 \cdots B_m$  ( $f : A_1 \cdots A_n \xrightarrow{c} B_1 \cdots B_m$ ).  $\text{Id} : A \xrightarrow{c} A$  denotes the *identity function* on  $A$ :  $\forall x : \text{Id}(x) = x$ .

For constructive functions  $f : A \xrightarrow{c} B$  and  $g : B \xrightarrow{c} C$ , their *serial composition*  $g \circ f$  is the normal function composition  $(g \circ f)(x) = g(f(x))$ . The *parallel composition* of  $f : A \xrightarrow{c} B$  and  $g : A' \xrightarrow{c} B'$  is denoted  $f \parallel g : A \cdot A' \xrightarrow{c} B \cdot B'$  and is defined by  $(f \parallel g)(x, y) = (f(x), g(y))$ . We assume that parallel composition operator binds stronger than serial composition, i.e.  $f \parallel g \circ h$  is the same as  $(f \parallel g) \circ h$ .

For a constructive function  $f : A \xrightarrow{c} A$  its least fixpoint always exists [9], and we use it to define a *feedback composition*. If  $f : A \cdot B \xrightarrow{c} A \cdot B'$  is a constructive function, then its feedback (on  $A$ ), denoted  $\text{feedback}(f) : B \xrightarrow{c} B'$ , is defined by

$$\text{feedback}(f)(y) = f(\mu x : f_1(x, y), y)$$

where  $f_1 : A \cdot B \xrightarrow{c} A$  is the first component of  $f$  and  $(\mu x : f_1(x, y))$  is the least fixpoint of the function that, for fixed  $y$ , maps  $x$  into  $f_1(x, y)$ .

Let  $x_1, \dots, x_n$  be variables ranging over types  $A_1, \dots, A_n$ , and  $e_1, \dots, e_m$  expressions using basic operations  $(+, -, \dots)$  on these variables, ranging over types  $B_1, \dots, B_m$ . We define the constructive function

$$[x_1, \dots, x_n \rightsquigarrow e_1, \dots, e_m] : A_1 \cdots A_n \xrightarrow{c} B_1 \cdots B_m$$

as the function that maps  $(x_1, \dots, x_n) \in A_1^\perp \times \dots \times A_n^\perp$  into  $(e_1, \dots, e_m)$ , where the basic operations are extended to unknown values in a standard way (e.g.  $3 + \perp = \perp$ ,  $\perp \cdot 0 = 0$ ).

## 4 Overview of the Translation Algorithm

A *block diagram*  $N$  is a network of interconnected blocks. A block may be a basic (*atomic*) block, or a *composite* block that corresponds to a *sub-diagram*. If  $N$  contains composite blocks then it is called a *hierarchical block diagram* (HBD); otherwise it is called *flat*. An example of a flat diagram is shown in Figure 1a. The connections between blocks are called wires, and they have a source block and a target block. For simplicity, we will assume that every wire has a single source and a single target. This can be achieved by adding extra blocks. For instance, the diagram of Figure 1a can be transformed as in Figure 1b by adding an explicit block called *Split*.

Let us explain the idea of the translation algorithm. We first explain the idea for flat diagrams, and then we extend it recursively for hierarchical diagrams.

A diagram is represented in the algorithm as a list of elements corresponding to the basic blocks. One element of this list is a triple containing a list of input variables, a list of output variables, and a *function*.

The function computes the values of the outputs based on the values of the inputs, and for now it can be thought of as a constructive function. Later this function will be an element of an abstract algebra modeling HBDs. Wires are represented by matching input/output variables from the block representations.

A block diagram may contain *stateful* blocks such as delays or integrators. We model these blocks using additional state variables (wires). In Figure 1, the only stateful block is the block Delay. This block is modeled as an element with two inputs  $(x, s)$ , two outputs  $(y, s')$  and function  $(y, s') := (s, x)$  (Figure 1c). More details about this representation can be found in [11].

In summary, the list representation of the example of Figure 1 is the following:

$$\begin{aligned} & [\text{Add}, \text{Delay}, \text{Split}] \text{ where} \\ & \text{Add} = ((u, z), x, [u, z \rightsquigarrow u + z]) \\ & \text{Delay} = ((x, s), (y, s'), [x, s \rightsquigarrow s, x]) \\ & \text{Split} = (y, (z, v), [y \rightsquigarrow y, y]) \end{aligned}$$

The algorithm works by choosing nondeterministically some elements from the list and replacing them with their appropriate composition (serial, parallel, or feedback). The composition must connect all the matching variables. Let us illustrate how the algorithm may proceed on the example of Figure 1; for the full description of the algorithm see Section 6.

Suppose the algorithm first chooses to compose Add and Delay. The only matching variable in this case is  $x$ , between the output of Add and the first input of Delay. The appropriate composition to use here is serial composition, but because Delay also has  $s$  as input, Add and Delay cannot be connected in series directly, as the number of outputs of Add would need to match the number of inputs of Delay. To achieve this, Add must first be composed in parallel with the identity block Id, as shown in Figure 1d. Doing so, a new element  $A$  is created:  $A = ((z, u, s), (y, s'), [x, s \rightsquigarrow s, x] \circ ([u, x \rightsquigarrow u + z \parallel \text{Id}])$ . Next,  $A$  is composed with Split. In this case we need to connect variable  $y$  (using serial composition), as well as  $z$  (using feedback composition). The resulting element is

$$((u, s), (v, s'), \text{feedback}([y \rightsquigarrow y, y] \parallel \text{Id}) \circ [x, s \rightsquigarrow s, x] \circ ([u, z \rightsquigarrow u + z] \parallel \text{Id}))$$

where we need again to add the Id component for variable  $s'$ .

Note that the final result of the algorithm is a triple with the same structure as all elements on the original list: (input variables, output variables, function), where the function represents the computation performed by the entire diagram. Therefore, the algorithm can be applied recursively on HBDs.

Also note that in this example the variables in the representation occur at most twice, once as input, and once as output. The variables occurring only as inputs are the inputs of the resulting final element, and variables occurring only as outputs are the outputs of the resulting final element. This is true in general for all diagrams, due to the representation of splitting of wires. This fact is essential for the correctness of the algorithm as we will see in Section 6.

## 5 An Abstract Algebra for Hierarchical Block Diagrams

We assume that we have a set of **Types**. We also assume a set of *diagrams* **Dgr**. Every element  $S \in \text{Dgr}$  has input type  $t \in \text{Types}^*$  and output type  $t' \in \text{Types}^*$ . If  $t = t_1 \cdots t_n$  and  $t' = t'_1 \cdots t'_m$ , then  $S$  takes as input a tuple of the type  $t_1 \times \dots \times t_n$  and produces as output a tuple of the type  $t'_1 \times \dots \times t'_m$ . We denote the fact that  $S$  has input type  $t \in \text{Types}^*$  and output type  $t' \in \text{Types}^*$  by  $S : t \xrightarrow{\circ} t'$ . The elements of **Dgr** are abstract.

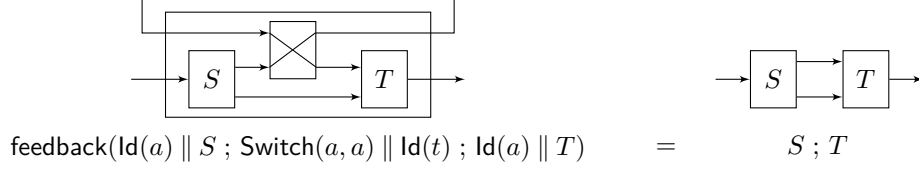


Figure 2: Two flat diagrams and their corresponding terms in the abstract algebra.

## 5.1 Operations of the HBDs Algebra

**Constants.** Basic blocks are modeled as constants on  $\text{Dgr}$ . For types  $t, t' \in \text{Types}^*$  we assume the following constants:

$$\begin{aligned} \text{Id}(t) &: t \xrightarrow{\circ} t \\ \text{Split}(t) &: t \xrightarrow{\circ} t \cdot t \\ \text{Sink}(t) &: t \xrightarrow{\circ} \epsilon \\ \text{Switch}(t, t') &: t \cdot t' \xrightarrow{\circ} t' \cdot t \end{aligned}$$

$\text{Id}$  corresponds to the identity block. It copies the input into the output. In the model of constructive functions  $\text{Id}(t)$  is the identity function.  $\text{Split}(t)$  takes an input  $x$  of type  $t$  and outputs  $x \cdot x$  of type  $t \cdot t$ .  $\text{Sink}(t)$  returns the empty tuple  $\epsilon$ , for any input  $x$  of type  $t$ .  $\text{Switch}(t, t')$  takes an input  $x \cdot x'$  with  $x$  of type  $t$  and  $x'$  of type  $t'$  and returns  $x' \cdot x$ . In the model of constructive functions these diagrams are total functions and they are defined as explained above. In the abstract model, the behaviors of these constants is defined with a set of axioms (see below).

**Composition operators.** For two diagrams  $S : t \xrightarrow{\circ} t'$  and  $S' : t' \xrightarrow{\circ} t''$ , their *serial composition*, denoted  $S ; S' : t \xrightarrow{\circ} t''$  is a diagram that takes inputs of type  $t$  and produces outputs of type  $t''$ . In the model of constructive functions, the serial composition corresponds to function composition ( $S ; S' = S' \circ S$ ). Please note that in the abstract model we write the serial composition as  $S ; S'$ , while in the model of constructive functions the first diagram that is applied to the input occurs second in the composition.

The *parallel composition* of two diagrams  $S : t \xrightarrow{\circ} t'$  and  $S' : r \xrightarrow{\circ} r'$ , denoted  $S \parallel S' : t \cdot r \xrightarrow{\circ} t' \cdot r'$ , is a diagram that takes as input tuples of type  $t \cdot r$  and produces as output tuples of type  $t' \cdot r'$ . This parallel composition corresponds to the parallel composition of constructive functions.

Finally we introduce a *feedback composition*. For  $S : a \cdot t \xrightarrow{\circ} a \cdot t'$ , where  $a \in \text{Types}$  is a single type, the feedback of  $S$ , denoted  $\text{feedback}(S) : t \xrightarrow{\circ} t'$ , is the result of connecting in feedback the first output of  $S$  to its first input. Again this feedback operation corresponds to the feedback of constructive functions.

We assume that parallel composition operator binds stronger than serial composition, i.e.  $S \parallel T ; R$  is the same as  $(S \parallel T) ; R$ .

Graphical diagrams can be represented as terms in the abstract algebra, as illustrated in Figure 2. This figure depicts two diagrams, and their corresponding algebra terms. As it turns out, these two diagrams are equivalent, in the sense that their corresponding algebra terms can be shown to be equal using the axioms presented below.

## 5.2 Axioms of the HBDs Algebra

In the abstract algebra, the behavior of the constants and composition operators is defined by a set of axioms, listed below:

1.  $S : t \xrightarrow{\circ} t' \implies \text{Id}(t) ; S = S ; \text{Id}(t') = S$
2.  $S : t_1 \xrightarrow{\circ} t_2 \wedge T : t_2 \xrightarrow{\circ} t_3 \wedge R : t_3 \xrightarrow{\circ} t_4 \implies S ; (T ; R) = (S ; T) ; R$
3.  $\text{Id}(\epsilon) \parallel S = S \parallel \text{Id}(\epsilon) = S$

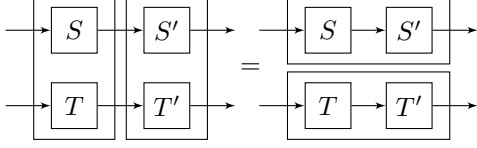


Figure 3: Axiom (5) Distributivity of serial and parallel.

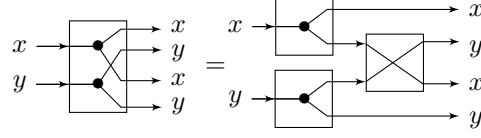


Figure 4: Axiom (11) Split switch.

4.  $S \parallel (T \parallel R) = (S \parallel T) \parallel R$
5.  $S : s \xrightarrow{\circ} s' \wedge S' : s' \xrightarrow{\circ} s'' \wedge T : t \xrightarrow{\circ} t' \wedge T' : t' \xrightarrow{\circ} t''$   
 $\implies (S \parallel T) ; (S' \parallel T') = (S ; S') \parallel (T ; T')$
6.  $\text{Split}(t) ; \text{Sink}(t) \parallel \text{Id}(t) = \text{Id}(t)$
7.  $\text{Split}(t) ; \text{Switch}(t, t) = \text{Split}(t)$
8.  $\text{Split}(t) ; \text{Id}(t) \parallel \text{Split}(t) = \text{Split}(t) ; \text{Split}(t) \parallel \text{Id}(t)$
9.  $\text{Switch}(t, t' \cdot t'') = \text{Switch}(t, t') \parallel \text{Id}(t'') ; \text{Id}(t') \parallel \text{Switch}(t, t'')$
10.  $\text{Sink}(t \cdot t') = \text{Sink}(t) \parallel \text{Sink}(t')$
11.  $\text{Split}(t \cdot t') = \text{Split}(t) \parallel \text{Split}(t') ; \text{Id}(t) \parallel \text{Switch}(t, t') \parallel \text{Id}(t')$
12.  $S : s \xrightarrow{\circ} s' \wedge T : t \xrightarrow{\circ} t' \implies \text{Switch}(s, t) ; T \parallel S ; \text{Switch}(t', s') = S \parallel T$
13.  $\text{feedback}(\text{Switch}(a, a)) = \text{Id}(a)$
14.  $S : a \cdot s \xrightarrow{\circ} a \cdot t \implies \text{feedback}(S \parallel T) = \text{feedback}(S) \parallel T$
15.  $S : a \cdot s \xrightarrow{\circ} a \cdot t \wedge A : s' \xrightarrow{\circ} s \wedge B : t \xrightarrow{\circ} t'$   
 $\implies \text{feedback}(\text{Id}(a) \parallel A ; S ; \text{Id}(a) \parallel B) = A ; \text{feedback}(S) ; B$
16.  $S : a \cdot b \cdot s \xrightarrow{\circ} a \cdot b \cdot t$   
 $\implies \text{feedback}^2(\text{Switch}(b, a) \parallel \text{Id}(s) ; S ; \text{Switch}(a, b) \parallel \text{Id}(t)) = \text{feedback}^2(S)$

Axioms (1) and (2) express the fact that the identity is neutral element for the serial composition, and serial composition is associative. Similarly, axioms (3) and (4) state that the identity of the empty type is neutral for the parallel composition, and that parallel composition is associative.

Axiom (5) introduces a distributivity property of serial and parallel compositions. Figure 3 represents graphically this axiom.

Axioms (6) - (11) express the properties of Split, Sink, and Switch. For example Axiom (11), represented in Figure 4, says that if we duplicate  $x \cdot y$  of type  $t \cdot t'$ , then this is equivalent to duplicate  $x$  and  $y$  in parallel, and then switch the middle  $x$  and  $y$ .

Axiom (12) says that switching the inputs and outputs of  $T \parallel S$  is equal to  $S \parallel T$ .

Axioms (13) - (16) are about the feedback operator. Axiom (13), represented in Figure 5, states that feedback of switch is identity. Axiom (14), represented in Figure 6, states that feedback of the parallel composition of  $S$  and  $T$  is the same as the parallel composition of the feedback of  $S$  and  $T$ . Axiom (15), Figure 7, states that components  $A$  and  $B$  can be taken out of the feedback operation. Finally, Axiom (16) represented in Figure 8, states that the order in which we apply the feedback operations does not change the result.

These axioms are equivalent to a subset of the axioms of algebra of flownomials [26], which implies that all models of flownomials are also models of our algebra. In [14] a relational model for dataflow is introduced.

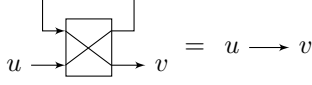


Figure 5: Axiom (13) Feedback of switch.

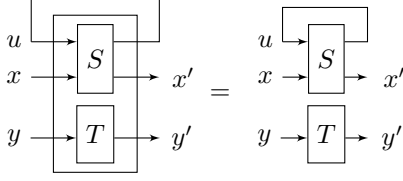


Figure 6: Axiom (14) Feedback of parallel.

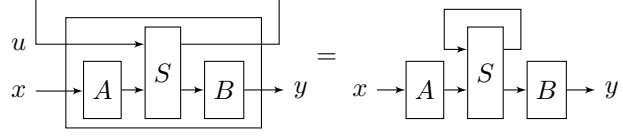


Figure 7: Axiom (15) Feedback of serial.

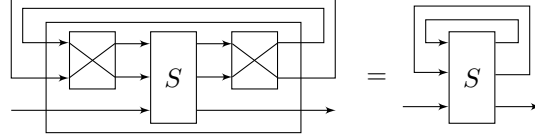


Figure 8: Axiom (16) Feedback of switched inputs/outputs.

This model is also based on a set of axioms on feedback, serial and parallel compositions, but [14] does not use the split constant. Our axioms that are not involving split are equivalent to the axioms used in [14]. The focus of [14] is the construction of a relational model for the axioms.

The following theorem provides a concrete semantic domain for HBDs.

**Theorem 1.** *Constructive functions with the operations defined in Section 3 are a model for axioms (1) – (16).*

All results in this paper have been formalized and proven in the Isabelle theorem prover, and are available at <http://rcrs.cs.aalto.fi/abstract-translation.zip>, together with several auxiliary results.

## 6 The Abstract Algorithm and its Correctness

### 6.1 Diagrams with Named Inputs and Outputs

The algorithm works by first transforming the graph of a HBD into a list of basic components with named inputs and outputs as explained in Section 4. For this purpose we assume a set of names or variables  $\text{Var}$  and a function  $T : \text{Var} \rightarrow \text{Types}$ . For  $v \in \text{Var}$ ,  $T(v)$  is the type of variable  $v$ . We extend  $T$  to lists of variables by  $T(v_1, \dots, v_n) = (T(v_1), \dots, T(v_n))$ .

**Definition 1.** A diagram with named inputs and outputs or *io-diagram* for short is a tuple  $(in, out, S)$  such that  $in, out \in \text{Var}^*$  are list of distinct variables, and  $S : T(in) \xrightarrow{\circ} T(out)$ .

In what follows we use the symbols  $A, A', B, \dots$  to denote io-diagrams, and  $I(A)$ ,  $O(A)$ , and  $D(A)$  to denote the input variables, the output variables, and the diagram of  $A$ , respectively.

**Definition 2.** For io-diagrams  $A$  and  $B$ , we define  $V(A, B) = O(A) \cap I(B) \in \text{Var}^*$ .

$V(A, B)$  is the list of common variables that are output of  $A$  and input of  $B$ , in the order occurring in  $O(A)$ . We use  $V(A, B)$  later to connect for example in series  $A$  and  $B$  on these common variables, as we did for constructing  $A$  from  $\text{Add}$  and  $\text{Delay}$  in Section 4.

### 6.2 General Switch Diagrams

We compose diagrams when their types are matching, and we compose io-diagrams based on matching names of input output variables. For example if we have two io-diagrams  $A$  and  $B$  with  $O(A) = u \cdot v$  and  $I(B) = v \cdot u$ , then we can compose in series  $A$  and  $B$  by switching the output of  $A$  and feeding it into  $B$ , i.e.,  $(A ; \text{Switch}(T(u), T(v)) ; B)$ .



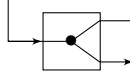


Figure 9: The diagram **Arb**.

In general, for two lists of variables  $x = (x_1 \cdots x_n)$  and  $y = (y_1 \cdots y_k)$  we define a *general switch diagram*  $[x_1 \cdots x_n \rightsquigarrow y_1 \cdots y_k] : T(x_1 \cdots x_n) \xrightarrow{\circ} T(y_1 \cdots y_k)$ . Intuitively this diagram takes as input a list of values of type  $T(x_1 \cdots x_n)$  and outputs a list of values of type  $T(y_1 \cdots y_k)$ , where the output value corresponding to variable  $y_j$  is equal to the value corresponding to the first  $x_i$  with  $x_i = y_j$  and it is arbitrary (unknown) if there is no such  $x_i$ . For example in the constructive functions model  $[u, v \rightsquigarrow v, u, w, u]$  for input  $(a, b)$  outputs  $(b, a, \perp, a)$ .

To define  $[\_ \rightsquigarrow \_]$  we use **Split**, **Sink**, and **Switch**, but we need also an additional diagram that outputs an arbitrary (or unknown) value for an empty input. For  $a \in \text{Types}$ , we define  $\text{Arb}(a) : \epsilon \xrightarrow{\circ} a$  by

$$\text{Arb}(t) = \text{feedback}(\text{Split}(a))$$

The diagram **Arb** is represented in Figure 9.

We define now  $[x \rightsquigarrow y] : T(x) \xrightarrow{\circ} T(y)$  in two steps. First for  $x \in \text{Var}^*$  and  $u \in \text{Var}$ , the diagram  $[x \rightsquigarrow u] : T(x) \xrightarrow{\circ} T(u)$ , for input  $a_1, \dots, a_n$  it outputs the value  $a_i$  where  $i$  is the first index such that  $x_i = u$ . Otherwise it outputs an arbitrary (unknown) value.

$$\begin{aligned} [\epsilon \rightsquigarrow u] &= \text{Arb}(T(u)) \\ [u \cdot x \rightsquigarrow u] &= \text{Id}(T(u)) \parallel \text{Sink}(T(x)) \\ [v \cdot x \rightsquigarrow u] &= \text{Sink}(T(v)) \parallel [x \rightsquigarrow u] \quad (\text{if } u \neq v) \end{aligned}$$

and

$$\begin{aligned} [x \rightsquigarrow \epsilon] &= \text{Sink}(T(x)) \\ [x \rightsquigarrow u \cdot y] &= \text{Split}(T(x)) ; ([x \rightsquigarrow u] \parallel [x \rightsquigarrow y]) \end{aligned}$$

### 6.3 Basic Operations of the Algorithm

The algorithm starts with a list of io-diagrams and repeatedly applies operations until it reduces the list to only one io-diagram. These operations are the extensions of serial, parallel and feedback from diagrams to io-diagrams.

**Definition 3.** The named serial composition of two io-diagrams  $A$  and  $B$ , denoted  $A ; B$  is defined by  $A ; B = (in, out, S)$ , where  $x = I(B) \ominus V(A, B)$ ,  $y = O(A) \ominus V(A, B)$ ,  $in = I(A) \oplus x$ ,  $out = y \cdot O(B)$  and

$$S = [in \rightsquigarrow I(A) \cdot x] ; D(A) \parallel [x \rightsquigarrow x] ; [O(A) \cdot x \rightsquigarrow y \cdot I(B)] ; [y \rightsquigarrow y] \parallel D(B)$$

The construction of  $A$  from Section 4 can be obtained by applying the named serial composition to **Add** and **Delay**.

Figure 10 represents an example of the named serial composition. In this case we have  $V(A, B) = u$ ,  $x = (a, b)$ ,  $y = (v, w)$ ,  $in = (a, c, b)$ , and  $out = (v, w, d, e)$ . The component  $A$  has outputs  $u, v, w$ , and  $u$  is also input of  $B$ . Variable  $u$  is the only variable that is output of  $A$  and input of  $B$ . Because the outputs  $v, w$  of  $A$  are not inputs of  $B$  they become outputs of  $A ; B$ . Variable  $a$  is input for both  $A$  and  $B$ , so in  $A ; B$  the value of  $a$  is split and fed into both  $A$  and  $B$ . The diagram for this example is:

$$[a, c, b \rightsquigarrow a, c, a, b] ; A \parallel \text{Id}(T(a, b)) ; [u, v, w, a, b \rightsquigarrow v, w, a, u, b] ; \text{Id}(T(v, w)) \parallel B.$$

The result of the named serial composition of two io-diagrams is not always an io-diagram. The problem is that the outputs of  $A ; B$  are not distinct in general. The next lemma gives sufficient conditions for  $A ; B$  to be an io-diagram.

**Lemma 1.** If  $A, B$  are io-diagrams and  $(O(A) - I(B)) \cap O(B) = \epsilon$  then  $A ; B$  is an io-diagram. In particular if  $O(A) \cap O(B) = \epsilon$  then  $A ; B$  is an io-diagram.



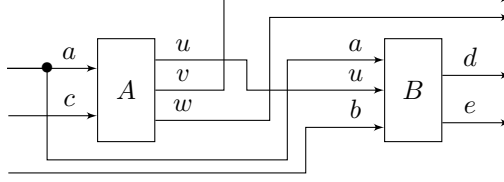


Figure 10: Example of named serial composition.

Next we introduce the corresponding operation on io-diagrams for the parallel composition.

**Definition 4.** If  $A, B$  are io-diagrams, then the named parallel composition of  $A$  and  $B$ , denoted  $A ||| B$  is defined by  $(I(A) \oplus I(B), O(A) \cdot O(B), S)$  where

$$S = [I(A) \oplus I(B) \rightsquigarrow I(A) \cdot I(B)] ; (A || B)$$

As in the case of named serial composition, the parallel composition of two io-diagrams is not always an io-diagram. Next lemma gives conditions for the parallel composition to be io-diagram and also states that the named parallel composition is associative.

**Lemma 2.** Let  $A, B$ , and  $C$  be io-diagrams, then

1.  $O(A) \cap O(B) = \epsilon \Rightarrow A ||| B$  is an io-diagram.
2.  $(A ||| B) ||| C = A ||| (B ||| C)$

Next definition introduces the feedback operator for io-diagrams.

**Definition 5.** If  $A$  is an io-diagram, then the named feedback of  $A$ , denoted  $FB(A)$  is defined by  $(in, out, S)$ , where  $in = I(A) \ominus V(A, A)$ ,  $out = O(A) \ominus V(A, A)$  and

$$S = \text{feedback}^{|V(A, A)|}([V(A, A) \cdot in \rightsquigarrow I(A)] ; S ; [O(A) \rightsquigarrow V(A, A) \cdot out])$$

The named feedback operation of  $A$  connects all inputs and outputs of  $A$  with the same name in feedback. The named feedback applied to an io-diagram is always an io-diagram.

**Lemma 3.** If  $A$  is an io-diagram then  $FB(A)$  is an io-diagram.

## 6.4 The Algorithm

We have now all elements for introducing the algorithm. The algorithm starts with a list  $\mathcal{A} = [A_1, A_2, \dots, A_n]$  of io-diagrams, such that for all  $i \neq j$ , the inputs and outputs of  $A_i$  and  $A_j$  are distinct ( $I(A_i) \cap I(A_j) = \epsilon$  and  $O(A_i) \cap O(A_j) = \epsilon$ ). We denote this property by  $\text{io-distinct}(\mathcal{A})$ . The algorithm is given in Alg. 1.

input:  $\mathcal{A} = [A_1, A_2, \dots, A_n]$  (list of io-diagrams)

while  $|\mathcal{A}| > 1$  :

choose:

- (a) choose  $k > 1$  and distinct  $B_1, \dots, B_k$  from  $\mathcal{A}$ :  
 $\mathcal{A} := [FB(B_1 ||| \dots ||| B_k)] \cdot (\mathcal{A} - [B_1, \dots, B_k])$
- (b) choose distinct  $A, B$  from  $\mathcal{A}$ :  
 $\mathcal{A} := [FB(FB(A)) ; ; FB(B)] \cdot (\mathcal{A} - [A, B])$

return  $FB(A)$  (where  $A$  is the only remaining element in  $\mathcal{A}$ )

**Alg. 1:** Nondeterministic algorithm for translating HBDs.

Computing  $\text{FB}(A)$  in the last step of the algorithm is necessary only if  $\mathcal{A}$  contains initially only one element. However, computing  $\text{FB}(A)$  always at the end does not change the result since, as we will see later in Theorem 2,  $\text{FB}$  operation is idempotent, i.e.  $\text{FB}(\text{FB}(A)) = \text{FB}(A)$ .

The result for the running example from Section 4 can be obtained by applying the second choice of the algorithm twice for the initial list of io-diagrams ( $[\text{Add}, \text{Delay}, \text{Split}]$ ), first to  $\text{Add}$  and  $\text{Delay}$  to obtain  $A$  and next to  $A$  and  $\text{Split}$ .

## 6.5 Correctness of the Algorithm

The result of the algorithm depends on how the nondeterministic choices are resolved. However, in all cases the final io-diagrams are equivalent modulo a permutation of the inputs and outputs. To prove this, we introduce the concept *io-equivalence* for two io-diagrams.

**Definition 6.** Two io-diagrams  $A, B$  are *io-equivalent*, denoted  $A \sim B$  if they are equal modulo a permutation of the inputs and outputs, i.e.,  $\text{I}(B)$  is a permutation of  $\text{I}(A)$ ,  $\text{O}(B)$  is a permutation of  $\text{O}(A)$  and

$$\text{D}(A) = [\text{I}(A) \rightsquigarrow \text{I}(B)] ; \text{D}(B) ; [\text{O}(B) \rightsquigarrow \text{O}(A)]$$

**Lemma 4.** *The relation io-equivalent is a congruence relation, i.e., for all  $A, B, C$  io-diagrams we have:*

1.  $A \sim A$ ,  $A \sim B \Rightarrow B \sim A$ , and  $A \sim B \wedge B \sim C \Rightarrow A \sim C$ .
2.  $A \sim B \Rightarrow \text{FB}(A) \sim \text{FB}(B)$ .
3.  $\text{O}(A) \cap \text{O}(B) = \epsilon \Rightarrow A ||| B \sim B ||| A$ .

To prove correctness of the algorithm we also need the following results:

**Theorem 2.** *If  $A, B$  are io-diagrams such that  $\text{I}(A) \cap \text{I}(B) = \epsilon$  and  $\text{O}(A) \cap \text{O}(B) = \epsilon$  then*

$$\text{FB}(A ||| B) = \text{FB}(\text{FB}(A) ; ; \text{FB}(B)) \quad \text{and} \quad \text{FB}(\text{FB}(A)) = \text{FB}(A).$$

The proof of Theorem 2 is quite involved and requires several properties of diagrams (see our Isabelle implementation for details).

We can now state and prove the main result of this paper, namely, correctness of Algorithm 1.

**Theorem 3.** *If  $\mathcal{A} = [A_1, A_2, \dots, A_n]$  is the initial list of io-diagrams satisfying  $\text{io-distinct}(\mathcal{A})$ , then Algorithm 1 terminates, and if  $A$  is the io-diagram returned by the algorithm, then*

$$A \sim \text{FB}(A_1 ||| \dots ||| A_n)$$

*Proof.* It is easy to see that the algorithm terminates because at each step, the size of the list  $\mathcal{A}$  decreases. To prove the correctness of the algorithm we need an invariant for the while loop statement. The invariant must be true at the beginning of the while loop, it must be preserved by the body of the while loop, and it must establish the final post-condition ( $A \sim \text{FB}(A_1 ||| \dots ||| A_n)$ ). If  $\mathcal{A}_0 = [A_1, \dots, A_n]$  is the initial list of the io-diagrams, and  $\mathcal{A} = [C_1, \dots, C_m]$  is the current list of io-diagrams, then the invariant is

$$\text{inv}(\mathcal{A}) = \text{io-distinct}(\mathcal{A}) \wedge \text{FB}(C_1 ||| \dots ||| C_m) \sim \text{FB}(A_1 ||| \dots ||| A_n)$$

Initially  $\text{inv}(\mathcal{A})$  is trivially true, and it also trivially establishes the final post-condition. We need to prove that both choices in the algorithm preserve the invariant.

$$\begin{aligned} \text{inv}(\mathcal{A}) \wedge k > 1 \wedge B_1, \dots, B_k \text{ are distinct elements of } \mathcal{A} \\ \Rightarrow \text{inv}([\text{FB}(B_1 ||| \dots ||| B_k)] \cdot (\mathcal{A} \ominus [B_1, \dots, B_k])) \end{aligned} \tag{1}$$

and

$$\begin{aligned} \text{inv}(\mathcal{A}) \wedge A, B \text{ are distinct elements of } \mathcal{A} \\ \Rightarrow \text{inv}([\text{FB}(\text{FB}(A)) ; ; \text{FB}(B)] \cdot (\mathcal{A} \ominus [A, B])) \end{aligned} \tag{2}$$

We first prove (1). Assume  $\mathcal{A} = [C_1, \dots, C_m]$  and  $inv(\mathcal{A}) = \text{io-distinct}(\mathcal{A}) \wedge \text{FB}(C_1 ||| \dots ||| C_m) \sim \text{FB}(A_1 ||| \dots ||| A_n)$ , and let  $D_1 = \text{FB}(B_1 ||| \dots ||| B_k)$ , and  $[D_2, \dots, D_u] = \mathcal{A} - [B_1, \dots, B_k]$ . It follows that  $\text{io-distinct}([D_1, \dots, D_u])$ . We prove now that  $\text{FB}(D_1 ||| \dots ||| D_u) \sim \text{FB}(A_1 ||| \dots ||| A_n)$ .

$$\begin{aligned}
& \text{FB}(D_1 ||| \dots ||| D_u) \\
= & \{ \text{Theorem 2 and } ||| \text{ is associative} \} \\
& \text{FB}(\text{FB}(D_1) ;; \text{FB}(D_2 ||| \dots ||| D_u)) \\
= & \{ \text{Definition of } D_1 \} \\
& \text{FB}(\text{FB}(\text{FB}(B_1 ||| \dots ||| B_k)) ;; \text{FB}(D_2 ||| \dots ||| D_u)) \\
= & \{ \text{Theorem 2} \} \\
& \text{FB}(\text{FB}(B_1 ||| \dots ||| B_k) ;; \text{FB}(D_2 ||| \dots ||| D_u)) \\
= & \{ \text{Theorem 2 and } ||| \text{ is associative} \} \\
& \text{FB}(B_1 ||| \dots ||| B_k ||| D_2 ||| \dots ||| D_u) \\
\sim & \{ \text{Lemma 4 and } \{B_1, \dots, B_k\} \cup \{D_2, \dots, D_u\} = \{C_1, \dots, C_m\} \} \\
& \text{FB}(C_1 ||| \dots ||| C_m) \\
\sim & \{ \text{Assumptions and Lemma 4} \} \\
& \text{FB}(A_1 ||| \dots ||| A_n)
\end{aligned}$$

Property (2) can be reduced to property (1) by applying Theorem 2. □

## 7 Application: Proving Equivalence of Two Translation Strategies

To demonstrate the usefulness of our framework, we return to our original motivation, namely, the open problem of how to prove equivalence of the translation strategies introduced in [11]. Each translation strategy of [11] can be seen as a determinization (or refinement) of the algorithm of Section 6. In what follows we use this to show equivalence of two of the translation strategies proposed in [11]: the *feedback-parallel* and the *incremental* translation strategies. The third strategy, *feedbackless*, is omitted due to lack of space. Equivalence holds for feedbackless too, and will be presented in an extended version of this paper.

The feedback-parallel strategy is the implementation of the abstract algorithm where we choose  $k = |\mathcal{A}|$ . Intuitively, all diagram components are put in parallel and the common inputs and outputs are connected via feedback operators. On the running example from Figure 1c, this strategy will generate the following component:

$$\begin{aligned}
& ((u, s), (v, s'), \text{feedback}^3([z, x, y, u, s \rightsquigarrow z, u, x, s, y] \\
& \quad ; \text{D(Add)} || \text{D(Delay)} || \text{D(Split)} ; [x, y, s', z, v \rightsquigarrow z, x, y, u, s']))
\end{aligned}$$

The switches are ordering the variables such that the feedback variables are first and in the same order in both input and output.

The incremental strategy is the implementation of the abstract algorithm where we use only the second choice of the algorithm and the first two components of the list  $\mathcal{A}$ . This strategy is dependent on the initial order of  $\mathcal{A}$ , and we order  $\mathcal{A}$  topologically (based on the input output connections) at the beginning, in order to reduce the number of switches needed.

Again on the running example, assume that this strategy composes first **Add** with **Delay**, and the result is composed with **Split**. The following component is then obtained:

$$((u, s), (v, s'), \text{feedback}(\text{D}(\text{Add}) \parallel \text{Id} ; \text{D}(\text{Delay}) ; \text{D}(\text{Split}) \parallel \text{Id}))$$

The **Add** and **Split** components are put in parallel with **Id** for the unconnected input and output state respectively. Next all components are connected in series with one **feedback** operator for the variable  $z$ .

The next theorem shows that the two strategies are equivalent, and they are independent of the initial order of  $\mathcal{A}$ .

**Theorem 4.** *If  $A$  and  $B$  are the result of the feedback-parallel and incremental strategies on  $\mathcal{A}$ , respectively, then  $A$  and  $B$  are input output equivalent ( $A \sim B$ ). Moreover both strategies are independent of the initial order of  $\mathcal{A}$ .*

*Proof.* Both strategies are refinements [4] of the nondeterministic algorithm. Therefore they satisfy the same correctness conditions (Theorem 3), i.e.

$$A \sim \text{FB}(A_1 \parallel \dots \parallel A_n) \text{ and } B \sim \text{FB}(A_1 \parallel \dots \parallel A_n)$$

where  $\mathcal{A} = [A_1, \dots, A_n]$ . From this, since  $\sim$  is transitive and symmetric, we obtain  $A \sim B$ .

For the second part, we use a similar reasoning. Let  $\mathcal{A} = [A_1, \dots, A_n]$ , and  $\mathcal{B} = [B_1, \dots, B_n]$  a permutation of  $\mathcal{A}$ . If  $A$  and  $B$  are the outputs of feedback-parallel on  $\mathcal{A}$  and  $\mathcal{B}$ , respectively, then we prove  $A \sim B$ . Using Theorem 3 again we have:

$$A \sim \text{FB}(A_1 \parallel \dots \parallel A_n) \text{ and } B \sim \text{FB}(B_1 \parallel \dots \parallel B_n).$$

Moreover, because  $\mathcal{B}$  is a permutation of  $\mathcal{A}$ , using Lemma 4 we have

$$\text{FB}(A_1 \parallel \dots \parallel A_n) \sim \text{FB}(B_1 \parallel \dots \parallel B_n).$$

Therefore  $A \sim B$ . The same holds for the incremental strategy. □

Since both strategies are refinements of the nondeterministic algorithm, they both satisfy the same correctness conditions of Theorem 3.

## 8 Conclusions and Future Work

We introduced an abstract algebra for hierarchical block diagrams, and a nondeterministic algorithm for translating HBDs to terms of this algebra. We proved that this algorithm is correct in the sense that no matter how the nondeterministic choices are resolved, the results are semantically equivalent. As an application, we closed a question left open in [11] by proving that the Simulink translation strategies presented there yield equivalent results. Our HBD algebra is reminiscent of the algebra of flownomials [26] but our axiomatization is more general, in the sense that our axioms are weaker. This implies that all models of flownomials are also models of our algebra. Here we presented constructive functions as one possible model of our algebra.

As future work we plan to present how the feedbackless translation strategy of [11] can be formalized in our framework and proved equivalent to the feedback-parallel and incremental strategies discussed above. We will also investigate alternative models for our abstract algebra, in addition to constructive functions. Such models include relations (which is a model in flownomials and thus also a model in our algebra), predicate transformers [10], and property transformers [20]. As discussed in [20], predicate and property transformers offer a more suitable framework for reasoning about reactive systems than functions and relations. However, there are subtleties in using such transformers as a model for the algebra of Section 5, since not all the axioms are always satisfied by that model. These subtleties have to do with the definition of instantaneous feedback [21] and will be further investigated in future work.

All our results have been formalized and proved in the Isabelle theorem prover and are available at <http://rcrs.cs.aalto.fi/abstract-translation.zip>.

**Acknowledgments.** We would like to thank Gheorghe Ștefănescu for his help with the algebra of flownomials.

## References

- [1] L. Ab. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015.
- [2] A. Agrawal, G. Simon, and G. Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004.
- [3] M. Amrani, B. Combemale, L. Lúcio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43, Aug. 2015.
- [4] R.-J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.
- [5] G. Berry. The Constructive Semantics of Pure Esterel, 1999.
- [6] O. Bouissou and A. Chapoutot. An Operational Semantics for Simulink’s Simulation Engine. *SIGPLAN Not.*, 47(5):129–138, June 2012.
- [7] D. Calegari and N. Szasz. Verification of model transformations. *Electronic Notes in Theoretical Computer Science*, 292:5 – 25, 2013.
- [8] C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
- [9] B. Davey and H. Priestley. *Introduction to lattices and order*. Cambridge University Press, New York, second edition, 2002.
- [10] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
- [11] I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional semantics and analysis of hierarchical block diagrams. In D. Bosnacki and A. Wijs, editors, *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, volume 9641 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2016.
- [12] Eclipse. ATL - a model transformation technology. <http://www.eclipse.org/at1/>.
- [13] S. Edwards and E. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comp. Progr.*, 48:21–42(22), July 2003.
- [14] T. T. Hildebrandt, P. Panangaden, and G. Winskel. A relational model of non-deterministic dataflow. *Mathematical Structures in Computer Science*, 14(5):613–649, 10 2004.
- [15] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer. Model transformation intents and their properties. *Softw. Syst. Model.*, 15(3):647–684, July 2016.
- [16] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.
- [17] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *ICFEM*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.
- [18] S. Minopoli and G. Frehse. SL2SX Translator: From Simulink to SpaceEx Models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC ’16*, pages 93–98, New York, NY, USA, 2016. ACM.

- [19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
- [20] V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *Embedded Software (EMSOFT)*. ACM, 2014.
- [21] V. Preoteasa and S. Tripakis. Towards compositional feedback in non-deterministic and non-input-receptive systems. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*, pages 768–777, New York, NY, USA, 2016. ACM.
- [22] R. Reicherdt and S. Glesner. Formal verification of discrete-time matlab/simulink models using boogie. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 190–204, Cham, 2014. Springer International Publishing.
- [23] M. Ryabtsev and O. Strichman. Translation validation: From simulink to c. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 696–701, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [24] S. Schlesinger, P. Herber, T. Göthel, and S. Glesner. Proving transformation correctness of refactorings for discrete and continuous simulink models. In *ICONS 2016, The Eleventh International Conference on Systems, EMBEDDED 2016, International Symposium on Advances in Embedded Systems and Applications*, pages 45–50. IARIA XPS Press, 2016.
- [25] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of simulink models into synchronous BIP. In *SIES*, pages 217–220, July 2010.
- [26] G. Stefanescu. *Network Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.
- [27] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.
- [28] C. Yang and V. Vyatkin. Transformation of Simulink models to IEC 61499 Function Blocks for verification of distributed control systems. *Control Engineering Practice*, 20(12):1259–1269, 2012.
- [29] C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.
- [30] L. Zou, N. Zhany, S. Wang, M. Franzle, and S. Qin. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *EMSOFT*, pages 9:1–9:10, Sept 2013.